Lecture 1: Introduction to Unix

References for Lecture 1:

1) Unix Network Programming, W.R. Stevens, 1990,Prentice-Hall, Chapter 1-2.
2) The C Programming Language, Kernighan and Ritchie, 2nd edition, Prentice-Hall, 1988
3) Unix Command: man !!!
4) Website for Unix Tutorial for Beginners: http://www.ee.surrey.ac.uk/Teaching/Unix/
5) Lecture notes are mainly for my teaching purpose, not as your only study material. You need to read books extensively.

Unix Commands:

Command format %command [options] [arguments]

1 **login**---most systems display a login prompt when you press a key. Type your user name and your password at the prompt

2 **man**--- give the user manual for Unix command, system calls, and C. **%man –s 2 write** for system call write not Unix command write. **%man write** for Unix command write. **%man –s 3c printf** for standard C library function.

3 **exit**---log off the system

4 **mkdir**---make a directory

5 **cd**---change the present directory, e.g. **%cd /usr/include %cd /etc/password**

6 **rmdir**---remove a directory. The directory must be empty.

7 **rm**---remore a file. Use **%rm –i** so it asks before deleting a file. Use **%rm –r** (recursive) to remove a directory and its contents, this is the most dangerous command.

8 **pwd**---print working directory.

9 **mv**---move (rename) a file.

10 **lp** or **lpr** ---line-print a file. Use –P to specify another printer.

11 **lpq**---query a print

12 **lprm**---rm a print job. Must be the user who printed the file.

13 **cat**---display the contents of a file

14 **more**--- display the contents of a file one screen at a time, SPACE CR, “q”, /pattern

15 **less**--- “less is more than more.” Try it and you will know why.

16 **head**---display the first ten lines. -# for more/less lines.

17 **tail**--- display the last ten lines. -# for more/less lines.

18 **cp**----copy a file. Mention * same for mv, etc.

19 **ls**---list files in current directory. Discuss –al.

20 **ps**---display a list of current processes on the system. Discuss –al.

21 **kill**---terminate a process. **%kill –9 PID**, must be owner of the process.

22 **grep**---search for a pattern in a file, etc. list all matching lines.

23 **clear**---clear the terminal window

24 **cal**---print calendar for current month.

25 **whereis**---explain what directory a command is stored in.

26 **chmod**—change permissions on a file/directory. 4=read 2=write 1=execute, absolute and relative: u+x o-r, g+w, etc.

27 **umask**-determine permissions of newly created files/directory. To keep your files from writing, chmod 755 * or umask 022.

28 **alias**—define new commands to simplify usage or modify existing commands, use unalias to undo.
%alias lpr "lpr –P cichp4k" %alias rm "rm –i" %alias dir "ls –al || lpr"
29 **history**---list the last 20 commands.
30 **vi**---full screen editor. Learn vi by using **% man vi** . Or other editors: emacs, vim, pico…
31 **find** ---find where a file is. **%find ./ -name a.out**
32 **uname**---What is the Unix version?
33 **hostname** --- host name of present computer
34 **ifconfig –a4** --- IP address of the host
35 **ping** --- check if a host is reachable. –a: get IP address of the host.

From the above discussion, we also want to introduce: full path name and relative path name, system file and user file, access permission mode, command options (they are optional and so called options, I think) .

Historical Stories: Why are Unix commands terse, and do not use the full English word? (e.g. use cp, mv unlike copy, move in DOS) But the lately added commands are quite long such as finger, hostname, history and so on. Why does kill use option –9?(Cats are said to be very difficult to kill because they have 9 lives. A persistent process may act like a cat. So you have to kill all 9 lives.)

Special characters:

1 !! — execute the last command again.
2 ! —!g: the latest command whose first letter is g, ghostview or gcc abc.c
3 !20 !?*word*
4 & — command can be run in the background, which disassociates them from the terminal and allow them to continue execution even after logging off.
5 < or > — open file as standard input, write standard output to file
6 >> — append standard output to the file.
7 << — here document. Used in scripts.
8 >! — ignor the effect of noclobber.
9 || — connection.

Setup files: .login .cshrc .profile .forward .signature

Such dotted files are called pretected or hidden files because they will not be found by file-matching metacharacter expansion.
When logging on, the **.login** and **.cshrc** files are executed. These files can be modified to customize your environment. Put your own alias, set(set local/shell variables) setenv(set global/environmental) variables. For example, setenv PRINTER cichp4k (makes cichp4k the default printer).

After changing the file, logout and log back in, or use
%source .cshrc

Questions: What is the difference between .login and .cshrc? There are a number of dotted files that end in the two-character sequence rc, meaning runnable commands. Depending which shell you are using, the setup file might be .profile (for Bourne shell) .bashrc (for BASH), tcshrc (for T shell), or others.

What is the difference between shell variables and environmental variables?

C language:

1 gcc — compiles C prgramm. Can slso use cc, but cc is platform-dependent.
Example: gcc –o abc –lsocket –lnsl a1.c a2.c ac.3
2 g++ — compiles C++ programs.

makefile:

Purpose: 1) define the dependency among different files, 2) used to compile programs without recompiling unchanged files. 3) %make -f filename --- if no filename is specified, makefile is the default name.

Format: target: components
|→ command 1
|→ command 2

Example: myprog: x.o y.o z.o
TAB gcc x.o y.o z.o -o myprog
x.o: x.c x.h
TAB gcc -c x.c # if x.c or x.h is new then recompile x.c
y.o: y.c x.h
TAB gcc –c y.c
z.o: z.c
TAB gcc –c z.c

Variables can be defined and used. This makes changes easy to implement. For example:

```
CC = gcc
CFLAGA = -O -c
LDFLAGES = -o
x.o: x.c x.h
TAB $(CC)  $(CFLAGS)  x.c
```

Argument list

```
main ( int argc, char *argv[ ])  /*  char *argv[] ==char **argv  */
  { }
```

e.g. %**echo hello world**: argc=3; arv[0]=echo; arv[1]=hello; arv[2]=world.

Environment list

```
main ( int argc, char *argv[ ], char *envp[]) /*environment list is optional, but main(char *envp[]) is wrong.*/
{   int i;
    for (i=0; envp[i] != (char *) 0; i++)
        printf("%s\n",envp[i]); /* print all environmental variable*/
    exit(0)
}
```

Can also use extern char **environ; to access variables. Use *getenv(char *val) to return the value of an environment variable, e.g.,

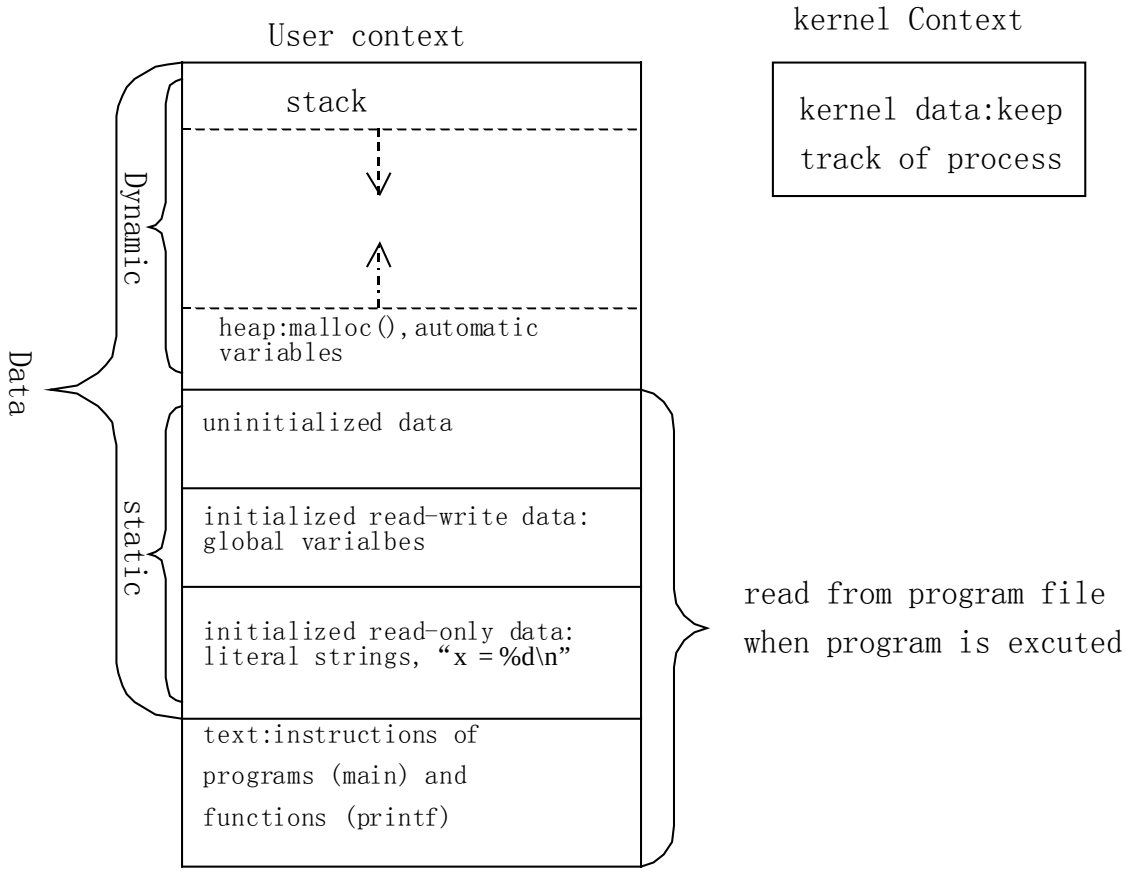
```
if ((ptr=getenv("HOME")) = = (char *) 0)
      printf("HOME is defined\n");
```

else printf("HOME=%s\n",ptr);

Program and Process

An instance of a **program** executed by the operating systems. Every Process has a unique process ID (PID). Parent process can only use the fork system call to create a new child process. It is interesting to further know the difference between process and thread; but it is beyond the scope of course.

Memory arrangement of process



Relationship between parent and child:

#include <unistd.h> /* all header files are under */user/include/* */
#include <sys/types.h> /* under */user/include/sys/* */
pid_t getpid(void), ----- returns process ID
pid_t getppid(void),------returns parent process ID
pid_t getpgid(void),------returns process group ID
uid_t getuid(void),------returns process user ID
gid_t getgid(void),------returns group ID

Process ID, parent process ID, process group ID, terminal group ID, user ID, group ID, effective user ID, effective group ID: So many ID, what are their purposes? Why are they necessary?

System Call

Unix Kernel (operating system) provides a limited number (60-200) of direct entry points through which an active process can obtain services from the kernel. These are named system calls.

1 Return value: –1 means: error occurs; >=0: OK.
2 Global interger value **errno:** contains the system error number, See <errno.h>,<sys/errno.h>.
3 When an error occurs, use perror( ) to print a message to explain the error.
4 Use **%man** to know more details.

SIGNALS

Singals are software interrupts and occur asynchronously at any time. For example, a segmentation fault produces a SIGSEGV signal. By default, SIGSEGV dumps core + terminate the process. SIGALRM signal is generated by using unsigned int alarm(unsigned int *seconds*). Default is to terminate the process.

1) Each signal is generated from **process→process** or from **kernel → process**.
2) Each signal has a name defined in "/usr/include/sys/iso/signal_iso.h", description and default action, e.g. SIGALRM; Alarm clock; Terminate. (Copy [Stevens90;P44] or <signal.h>).

How to generate signals:

process→process:

1) System call: int kill(int *pid*, int *sig*). It is because signals can be produced by program using kill( ) that signals are also called as software interrupts. kill( ) is not only used to kill a process, but also used in most cases to send a signal *sig* to a process *pid*. (Copy [Stevens90;P45]).
2) Command: kill –9 *pid*, Generate SIGTERM. Don't confuse kill command with system call of the same name.

kernel → process:

3) Keyboard: control-C or Delete→SIGINT. Control-backslash→SIGQUIT. Control-Z→SIGTSTP.
4) Hardware: SIGSEGV or SIGFPE for Floating Point Arithmetic errors.
5) Software condition: Special software conditions are noticed by kernel such as SIGURG for out-of-band data on a socket.

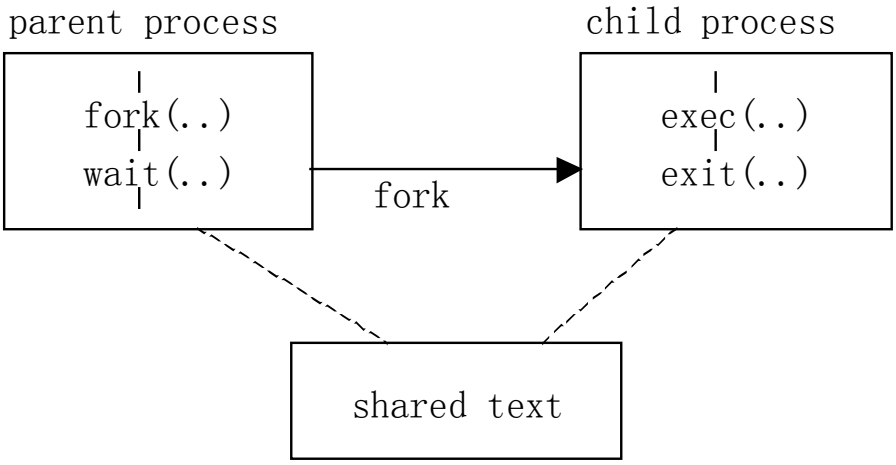
How to handle signals:

1) #include <signal.h>
   int (*signal (int *sig*, void (**func*)(int))) (int);
   When *sig* occurs, *func*, signal handler, is called to handle the condition.
2) 2 special values for the *func* argument: SIG_DFL: perform the default action, e.g. signal(SIGALRM, SIG_DFL). SIG_IGN: ignore the signal except SIGKILL and SIGSTOP.
3) block and unblock a signal: See a more detailed example for asynchronous I/O in Lecture

```
oldmask=sigblock(sigmask(SIGINT) | sigmask(SIGQUIT));
/* critical region */
sigsetmask(oldmask);
```

Process Control

int fork() : creates a copy of the current process called the child process. They may share the same text segment.



fork( ) is called once, but returns twice:
-1: if an error occurred
0:returned to the child process
PID: returned to the parent process
The child process inherits but cannot influence the environment of the parent process.

int exec( ): the child process can then use the exec( ) to start a new program. Every process in the Unix except the first process is created by the combined efforts of fork() and exec().

int exit(int *status*): child process finishes by passing status to kernel. Kernel can transfer status to parent which is executing wait( ).

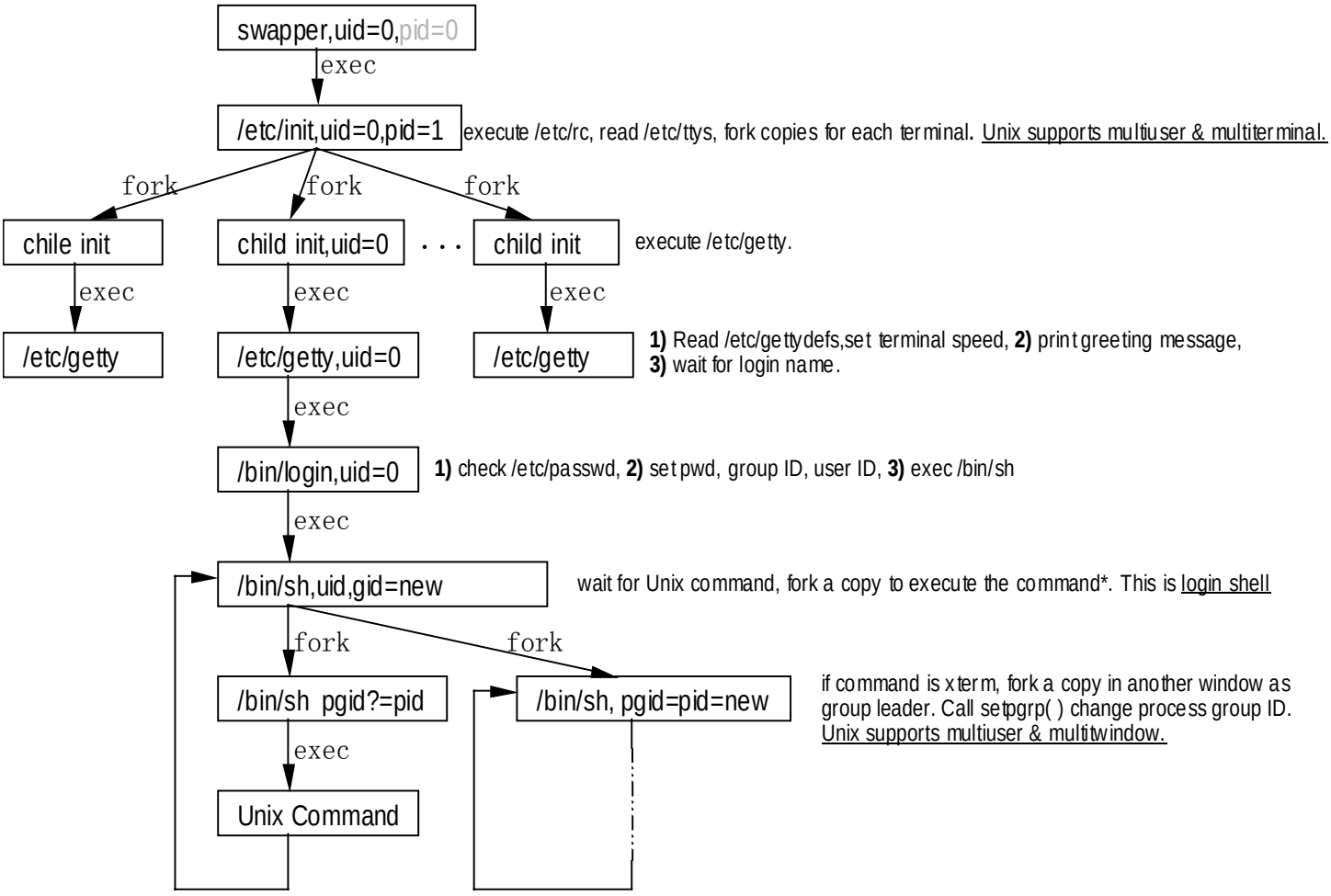
int wait(int ****status***):** the parent process can **wait(…)** for the child process to finish. Otherwise if parent process doesn't execute wait(…), the child will become a zombie process when the child process ends with **exit(…)**[①]. But a zombie process will last only until the parent process terminates[②]. If the parent process ends first, the parent process ID of the child (an orphan now) is set to 1. **wait(*status*)** returns the process ID of the child process[③]. ***status*** gets the *status* value of exit(*status*) of the terminating child process.

NOTE: ① It has been tested on my computer that the child will become a zombie in this case even if it has no exit(…) as the last line. ② A daemon process (running for ever) should call signal(SIGCLD,SIG_IGN) to avoid zombies. ③ If one parent

forks many children, the parent's wait(status) will get the status of the first finished child. All the remaining children will become zombies until the parent terminates.

Homework: learn how to use the following system calls.
stat(), fstat(),dup(), dup2(),fcntl(), ioctl(), execl(); execle( ), execlp( ), execv( ), execve( ) and execvp().

Process Relationships:



***** Unix commands are either part of the shell (internal commands), or they are stored on disk and must be brought into memory before execution (external commands). , Internal commands, also called as built-in commands or kernel space functions, run within the current shell while external commands require a new shell be created for them to run in. Use **%man shell_builtins** to list all the built-in commands on your system.

Shell: interface between user and OS to accept your command, send to OS for execution and return the result.

More than a command interpreter, it is a programming language.

Comparison of different Shells:

	Bourne Shell	**C Shell**	**Korn Shell**	**BASH**
Command	sh	csh	ksh	bash
Prompt	%	$	$	$
Author	Ken Thompson and Steven Bourne(Bell Lab)	Bill Joy UC Berkeley	David Korn Bell Lab	Borne Again SHell OSF, = csh + ksh
Redirection of Error	cmd 1>outfile 2> errfile cmd 2> &1	cmd>&outfile (cmd<infile>outfile)>&errfile	Same as bsh	
variable	**local**: %name=value **global**:%export name	**local**:%set name=value %@name=value **global**:%setenv name value	Same as bsh	
Job control	no	Ctrl-Z,jobs,bg,fg stop	Same as csh	
History	no	history, !!,!?word, !20	Better than csh	
Alias	no	Alias name string	Alias name=string	
Notes:	First shell, very fast	C-like syntax		Default shell for Linux; backward compatible with BSH

- How to know which shell you are using(use ps)? How to change Shell? What is difference between local variable and global variable?
- The effect of setenv or unsetenv can only been seen by all the shells forked by the present shell

Daemon Process:

Runs in the background and wait for an event to occur or executes on a periodic basis. Examples include: mail daemon, printer daemon, etc.

How to start a daemon process:

For system daemons, they are put in shell script /etc/rc and will be executed by /etc/init. For user's daemons, they can be put into .cshrc or at the command line appended by &.

Rules to follow for creating a robust daemon:

- Close all open file descriptor including stdin, stdout, stderr
- Change current working directory—this allows the sysadmin to unmount a file system without killing the daemon.
- Reset the file access creation mask—this way file created by the daemon have the correct protections
- Run in the background—fork a child process + exit from the parent process
- Disassociate from the process group—prevent the daemon from being affected by signals sent to the process group
- Ignore terminal I/O signals—allow the daemon process from receiving terminal signals
- The cases above might not occur in all daemons.

A Skeleton of Daemon Code: (This program involves all the topics discussed in Lecture 1.)

```
/** Initialize a daemon process. */
#include<stdio.h>
#include<signal.h>
#include<sys/param.h>
#include<errno.h>
extern int      errno;

#ifdef     SIGTSTP          /* true if BSD system */
#include<sys/file.h>
#include<sys/ioctl.h>
#endif

/* * Detach a daemon process from login session context.    */

daemon_start(ignsigcld)
int   ignsigcld;     /* nonzero -> handle SIGCLDs so zombies don't clog */
{
      register int    childpid, fd;

/* If we were started by init (process 1) from the /etc/inittab file there's no need to detach. This test is unreliable due to an unavoidable ambiguity if the process is started by some other process and orphaned (i.e., if the parent process terminates before we are started). */

      if (getppid() == 1)
```

```
        goto out;

/** Ignore the terminal stop signals (BSD). */

#ifdef SIGTTOU
    signal(SIGTTOU, SIG_IGN);
#endif
#ifdef SIGTTIN
    signal(SIGTTIN, SIG_IGN);
#endif
#ifdef SIGTSTP
    signal(SIGTSTP, SIG_IGN);
#endif

/* If we were not started in the background, fork and let the parent exit.  This also guarantees the first child is not a process group leader. */

    if ( (childpid = fork()) < 0)
        err_sys("can't fork first child");
    else if (childpid > 0)
        exit(0);  /* parent */

/** First child process.

Disassociate from controlling terminal and process group. Ensure the process can't reacquire a new controlling terminal. */

#ifdef  SIGTSTP         /* BSD */

    if (setpgrp(0, getpid()) == -1)
        err_sys("can't change process group");

    if ( (fd = open("/dev/tty", O_RDWR)) >= 0) {
        ioctl(fd, TIOCNOTTY, (char *)NULL); /* lose controlling tty */
        close(fd);
    }

#else   /* System V */

    if (setpgrp() == -1)
        err_sys("can't change process group");

    signal(SIGHUP, SIG_IGN);    /* immune from pgrp leader death */
```

```
        if ( (childpid = fork()) < 0)
                err_sys("can't fork second child");
        else if (childpid > 0)
                exit(0);  /* first child */

        /* second child */
#endif

out:
        /*
         * Close any open files descriptors.
         */

        for (fd = 0; fd < NOFILE; fd++)
                close(fd);

        errno = 0;              /* probably got set to EBADF from a close */

/* Move the current directory to root, to make sure we aren't on a mounted file system.    */

        chdir("/");

/* Clear any inherited file mode creation mask. */

        umask(0);

/* See if the caller isn't interested in the exit status of its children, and doesn't want to have them become zombies
and clog up the system.    With System V all we need do is ignore the signal. With BSD, however, we have to
catch each signal and execute the wait3() system call. */

        if (ignsigcld) {
#ifdef  SIGTSTP
                int   sig_child();

                signal(SIGCLD, sig_child);      /* BSD */
#else
                signal(SIGCLD, SIG_IGN);    /* System V */
#endif
        }
}
```